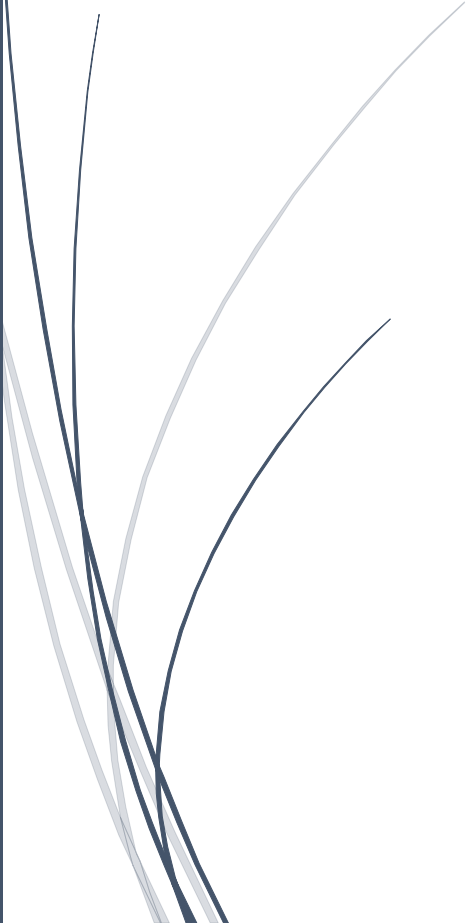


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

26-1-2020

# FINAL TASK EXAM

Reinforcement Learning with Pac-Man

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Luis Ríos, Jorge Chueca & Javier Verón  
INTELLIGENT SYSTEMS, SAN JORGE UNIVERSITY

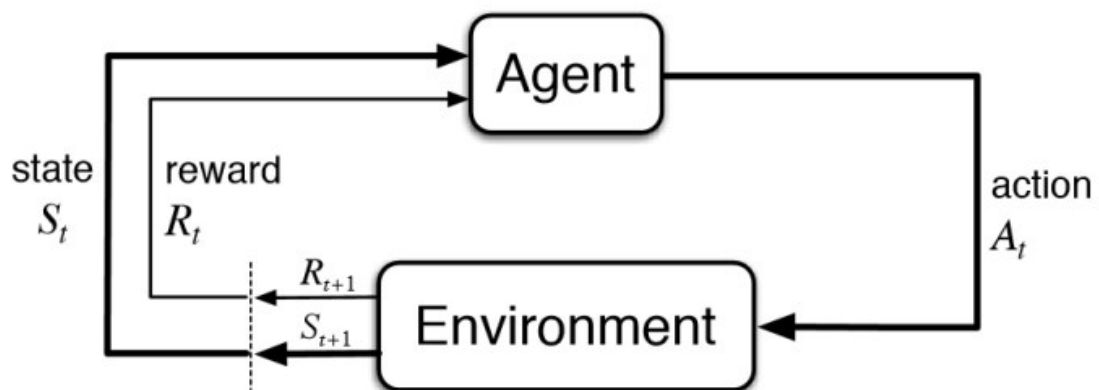
---

## FIRST PART: PRACTICAL PART

---

### Reinforcement Learning<sup>[1]</sup>

Since videogames are considered virtual worlds, we use reinforcement learning to help our AI to learn how to best interact with its environment. Time delayed labels called **rewards** are fed to the agent (Pacman in our case) in a process of trial and error to help it learn how best to act in the environment. If the agent makes a mistake a negative reward will be given, if it makes a good action a positive reward will be given. Iterating through this process the agent learns which actions are the optimal to perform in a given situation. It is easily explained by the next diagram:



Here we can see that in a given state the agent performs an action in the environment, it then collects a reward and moves to the next state. We need some stop conditions, in our case these are: dying losing 3 lives, eating all the apples or a time limit. For example, eating an apple will give a positive reward and dying or the time passing will give a negative reward.

Each game played the actions to perform are updated for the next game, giving conservation and continuity in the learning process. This way the agent can learn from its mistakes and/or successes.

Reinforcement learning is a **model-free**<sup>[2]</sup> type of AI, this means that the AI can directly derive an optimal policy from its interactions with the environment without needing to create a model beforehand. At the start of the training the agent doesn't know the environment and it needs to explore it and interact with it in order to perform as much actions as possible in the different states. This way it has a base of knowledge where it can decide between more actions finding the best possible policy.

From this appears the **exploration vs exploitation dilemma**<sup>[1]</sup>, about whether the AI should trust the learnt actions enough (exploitation) or try other actions hoping that it might give a better reward (exploration). Both extremes are not desirable, if the agent always explores it does not make any

decision, it is just wondering randomly through the environment and if the agent always exploits it does not perceive the environment well enough to make a solid decision. Usually, the agent starts exploring a lot and the chance of exploring keeps decreasing through time as it knows more about its surroundings, but there is always some small chance to choose to explore in order to give more richness of decision and pushing the agent towards the best optimal policy.

## Algorithms

### Monte Carlo<sup>[3]</sup>

Monte Carlo (MC) methods are a subset of computational algorithms that use the process of repeated random sampling to make numerical estimations of unknown parameters. They allow for the modelling of complex situations where many random variables are involved.

The way it works is by doing random games. Once it finishes it calculates the mean reward of each state it has been for this run and the ones it has done before. This is just the training method, after a couple trains we have a map of the rewards in each state.

0.73	0.81	0.9	1.00
0.66		-0.9	-1.00
0.59	??	-0.81	-0.9

There are a couple of disadvantages for this method, one of them is that you can't update the mean reward of a state until you finish the game, some games may be infinite.

Also, this method doesn't assure you that you will visit all the states.

## TD (n) <sup>[4]</sup>

Temporal Difference works in a similar way to how MC works, the main difference between them is that TD can update the mean of the reward of a state after leaving that state. A good example to see this method is a GPS, where you have an estimation of time when you want to arrive to a place, if you go faster the time will be reduced in real time. If after that you go slower, the time will be increased. On the other hand, MC will tell all the information of the trip once you have arrived at the place.

The TD method has an input value, that oversees looking a number of steps in the future. So, if this number is TD (1) it will only look forward one step and will only know the final reward when you are in that step.

## Q-Learning <sup>[5]</sup> <sup>[6]</sup>

Q-Learning is an algorithm in Reinforcement Learning for the purpose of strategy learning.

Mathematically, the strategy in Q-Learning is modelled as an action-value function  $Q(s, a)$ , which represents how much is the Agent willing to take an action 'a' in given state 's'.

In addition, the value of  $Q(s, a)$  returned is called Q-value. The larger Q-value, the larger possibility of the action a will be taken since the Agent expects to get more rewards if it has large Q-value.

We use  $Q(s, a)$  as the policy of the Agent, so if the best  $Q(s, a)$  is found, then the Agent has learned the policy perfectly. Therefore, our goal here is to find the best  $Q(s, a)$ .

The Cost Function (or Loss Function) is used for measuring the quality in Supervised Learning and it can be calculated precisely since the label (i.e. right answer) is known.

It is also used in Reinforcement Learning, but the right answer is always unknown. Then, how do we estimate the quality of the policy?

$$\text{LossFunction} = \boxed{Q_{best}(s_t, a_t)} - \boxed{Q(s_t, a_t)}$$

TD-target is UNKNOWN!      Current Q-value

The way we apply to estimate the best  $Q(s, a)$  is Bellman Equation

$$\text{LossFunction} = \boxed{R_{t+1} + \gamma \max_a Q(s_{t+1}, a)} - \boxed{Q(s_t, a_t)}$$

Estimated TD-target      Discount factor      Reward      Maximum next-state Q-value      Current Q-value

We want to minimize Loss Function using, for example, Gradient Descent:

$$\begin{array}{c}
 \text{Update Current Q-value} \\
 \boxed{Q(s_t, a_t)} \leftarrow \boxed{Q(s_t, a_t)} + \boxed{\alpha} (\boxed{R_{t+1}} + \boxed{\gamma} \boxed{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})} - \boxed{Q(s_t, a_t)}) \\
 \text{Current Q-value} \quad \text{Learning Rate} \quad \text{Rewards} \quad \text{Discount factor} \quad \text{Estimated reward from next action}
 \end{array}$$

Learning Rate is a hyper-parameter for controlling the convergent speed of updating procedure and Discount Factor is another hyper-parameter for weighting the importance of estimated future reward. The value of discount factor is between 0 and 1. The closer to 1, the more important the future reward is.

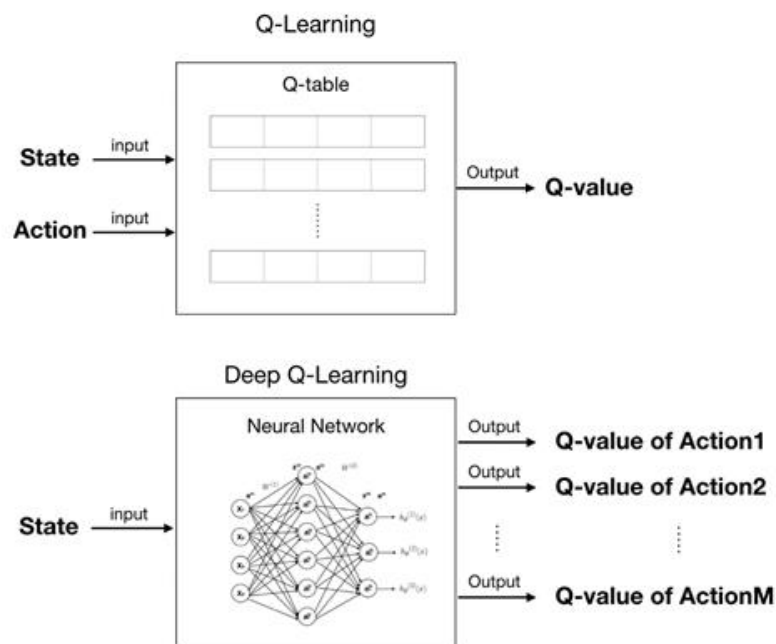
We will end up with a matrix to store what we know after some random values.

The problem with this is that the AI will never explore new actions because it knows the rewards of the already known ones.

The way to solve this problem is with random exploration. The IA will take eventually a random action so in the long term it will know more possibilities.

The explorations vs exploitation dilemma is solved then with an Epsilon variable that goes from 1 to 0.

Now we have another problem: with the matrix method, the space we need is n states by m actions. When these numbers are large such as 1000 states and 1000 actions per state, we would need a matrix of 1 million elements. Thus, the matrix method doesn't work in most interesting problems such as chess, which needs  $10^{120}$  states space. Instead of recording Q-value precisely with matrix, the approximately way is more feasible such as Neural Network (NN) framework. That is why we go deep:



We only must initialize hyper-parameters and the NN Model as we saw in this subject and, then, make it start training.

This might seem enough, but the basic Q-Learning algorithm has some disadvantages like overestimations of action value: Q-value, in basic (Deep) Q-Learning and introduce one popular solution, called Double Deep Q Network (Double DQN or DDQN), a Double Q-Learning with Neural Network architecture.

## Double Q-Learning <sup>[6] [7]</sup>

Briefly, the problem of overestimations is that the agent always chooses the non-optimal action in any given state only because it has the maximum Q-value.

The assumption behind the idea is that the best action has the maximum expected/estimated Q-value.

However, the Agent knows nothing about the environment in the beginning, it needs to estimate  $Q(s, a)$  at first and update them at each iteration. Such Q-values have lots of noises and we are never sure whether the action with maximum expected/estimated Q-value is really the best one. Therefore, the learning process will be very complicated and messy.

Double Q-Learning uses two different action-value functions,  $Q$  and  $Q'$ , as estimators. Even if  $Q$  and  $Q'$  are noisy, these noises can be viewed as uniform distribution.

The update procedure is slightly different from the basic version:

### Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

### Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \underbrace{Q'(s_{t+1}, \boxed{a})}_{\text{estimated/expected Q-value}} - Q(s_t, a_t))$$

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = \underbrace{Q'(s_{t+1}, \boxed{a})}_{\text{estimated/expected Q-value}}$$

$Q$  function is for selecting the best action 'a' with maximum Q-value of next state.

$Q'$  function is for calculating expected Q-value by using the action 'a' selected above.

Update  $Q$  function by using the expected Q-value of  $Q'$  function and we have it.

## Code and features

We tried to make the whole code in Python-Notebook (.ipynb) using Google Colab, but after some days of having problems we decided to try in Linux with everything done in Python files (.py) and run from the console. After one day of trying and not being able to make it fully work, we decided to use a Python-Notebook which uses Python files. That is the reason why you must open PacMan\_ReinforcementLearning.ipynb and call from there main.py.

Our teacher Rafael showed us how to call console commands like that one from a Python-Notebook correctly and, thus, we could do this.

The base from the Python code was provided by TowardsDataScience<sup>[6]</sup>.

When you call main.py, it expects some flags in the calling: The network you want to use specifying the algorithm (-n), the mode of execution you want (-m) and the path where you are going to save the results (-s). The saved data was required in previous versions using -l, but now we do it automatically depending on the network you want to use. This can be easily changed from the function "load\_network(self, path)" in deep\_Q.py and duel\_Q.py.

For more information about the execution of the program you can read the README.txt file.

The file main.py creates the game instance and loads the network in it. Then it just executes the methods it needs to make the program work, which are implemented in pacman.py and the two Q-Learning networks. In main.py you can also modify an hyperparameter called NUM\_FRAME which is the number of the frame when the execution will stop when training (that can be lots of lives, not just 3 as in the testing mode). The rest of hyperparameters are specified in pacman.py. Here you can change MIN\_OBSERVATION for specifying in which frame you want to start decreasing the Epsilon (for the exploration vs exploitation dilemma explained above) or the initial and final Epsilon, for example. The class ReplayBuffer creates buffer object that stores the past moves and samples a set of subsamples.

Below that, it is specified the number of frames the network will receive. It needs 3, not just 1, as it needs to know the movement of everything in the screen. With just one it wouldn't know if the enemies are going towards you or the opposite way, for example.

Then, in the init, it creates the appropriate network (based on the flags you entered before) and the buffer for the frames.

Then it has the methods for training, simulating and calculating mean. In training it is important that we save the network every 100000 iterations just in case there is a problem with the execution, and it gets stopped not to lose all the training data.

The deep\_Q.py and duel\_Q.py are very similar at a high level. The init function creates the neural networks with their own layers and features that we will explain later. We have save and load functions to have persistency in our networks. The function predict\_movement chooses the final action to perform taking into account the epsilon for the exploration vs exploitation dilemma. And

finally, they have a train function that updates the Q-values in the networks according with their own algorithm (double-Q and Q learning respectively).



---

## SECOND PART: SEARCH AND PLANNING

---

### TABU algorithm <sup>[8]</sup>

It is a type of metaheuristic search that uses local or neighbourhood search methods. These iterations change the current solution to an improved one, which is on the neighbourhood of the current solution.

This may create some problems, like getting stuck in a zone of poor scoring, but this can be avoided as TABU explores other possibilities, other solutions. To solve this problem TABU has a memory structure that determines all the possible neighbours that could be accepted so now he can improve the current solution.

This memory structures are called TABU list, that is form by a couple of rules used to choose the solutions that will be admitted as part of the neighbourhood. It is common to see solutions that change after you iterate in this algorithm. In the end there is a stop criterion that will determine when the search is done. The name of the algorithm comes from the solutions that are forbidden or not accepted.

Related to the salesman problem, one of the advantages that TABU searches offers is that it can avoid getting stuck when a set of cities are commonly repeated. On the other hand, if you don't understand the solution it will be difficult to set your list and your solution won't be optimal.

### Genetic algorithm <sup>[9]</sup>

It is also a type of metaheuristic search; it is all based in the natural selection and how is evolution conceived now a days. The algorithm hide under it relays on bio-inspired operations like mutations, crossover and natural selection.

It basically works as follows, there is a population of candidate solutions that will evolve to a better solution than the current, and in order to do that they have a set of properties that they can mutate and change.

At the beginning everyone from the population has been randomly created and once the iterations start, also called generations, the result gets better and better, they have an individual value that try to maximize, commonly called fitness. Those that got the best result will be used to get their genomes and create a new generation that will iterate and iterate.

Related to the salesman problem, it is usually an expensive algorithm. Also, the bigger the problem is the more expensive the algorithm will be due to the high number of mutations that can be created each generation. This algorithm has no stop criterion. It can get stuck in local optimal solutions.

## A\* algorithm <sup>[10]</sup>

Known as A-star is a graph traversal and path search algorithm. Used in computer science because it is complete and optimal. It makes use of weighted graphs. Basically, what it tries is to get to a certain node by having the smallest cost possible.

In order to accomplish this task, it is based in tree paths. Each node has a weight and each node also may have sons. On each iteration, it determines the path that is going to follow, and this path will be the result of adding the next node and the total cost that has already gone through.

The typical implementation is the following one: *“At each step of the algorithm, the node with the lowest  $f(x)$  value is removed from the queue, the  $f$  and  $g$  values of its neighbours are updated accordingly, and these neighbours are added to the queue. The algorithm continues until a goal node has a lower  $f$  value than any node in the queue (or until the queue is empty).”* Where  $f$  for the goal is the shortest path possible to be created.

Once you know the path you can storage the information of all the nodes of that path by keeping track of the predecessor node in the next one.

Related to the salesman problem, this algorithm seems like the one made for it, it seems like it was design as a pathfinding algorithm. If you set the distances as weight the only thing you can do is wait for the result. Depending on the complexity of the problem this algorithm can handle better than the TABU algorithm. And this one will never get stuck like the genetic one or the TABU if you don't know how to set the rules for the list.

## Overfitting <sup>[11][12][13]</sup>

When we talk about overfitting, we are talking about a common mistake done while training an AI. The main problem occurs when the result of that training is too exact and corresponds closely to a unique set of data. This will certainly cause a fail when trying to fit a new set of data with the knowledge learned with the previous data. Any way as you may have this problem you can also have the underfitting problem, which is less common but still possible if you miss the parameters that should be in a correct model.

In order to solve this problem in Keras there are different approaches, but the best one is trying first to train as much as possible your AI, as the more it learns by training, the more knowledge it will

have and overfitting or underfitting won't appear. If this first approach isn't possible or if you want to be more accurate you can implement constraints on the quality and type of information you obtain from your model.

There are different ways to implement constraints, one of them is weight regulation, that considers the fact that the explanation most likely to be correct is the "simplest" one.

It works as follows: *"A 'simple model' in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters altogether, as we saw in the section above). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to only take on small values, which makes the distribution of weight values more 'regular'."* <sup>[11]</sup>

## Example of backward chaining

Backward chaining is used in games like chess, where you have the goal you want to achieve, and you can go backwards to the current state of the game. In chess this method is called retrograde analysis<sup>[14]</sup> and it creates table bases.

Also, backward chaining is used in real life, for example when babies learn how to eat, they know that the goal is to have the food in the mouth and from there they go back one step at the time.

## Traveling salesman problem

Based on what we have learned in this second part, the optimal solution will probably be the A\* algorithm, which is critical for path finding and is the perfect approach for this problem, since each travel has his own weight so the A\* tree can be created really easy.

Another possible answer is using TABU, as this question suggests, but it will be a bad choice because the bigger the problem is the better A\* is compared to TABU. Also, TABU may be stuck in suboptimal solutions.

In order to be implemented, TABU should have a couple restrictions related to looping between cities, also it will have a restriction related to go to the lowest weight possible. Having some sort of exploration restriction to try to find new solutions or sub solutions that may be more optimal in long term.

---

## EXTRA: QUESTIONS ASKED ON THE ORAL EXPOSITION

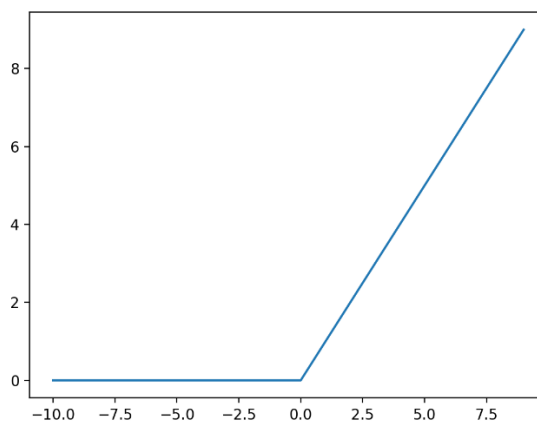
---

### What does ReLU mean

It means “Rectified Linear Unit” and, in neural networks, it is used to define those units that use the rectifier. A rectifier is a function defined as  $f(x)=x^+=\max(0, x)$ , where  $x$  is the input that the neuron receives.<sup>[15]</sup>

The function above means that the rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

Some other functions, as the sigmoid and hyperbolic tangent activation ones, cannot be used in networks with many layers due to the vanishing gradient problem. The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better. This way, the rectified linear activation is the default activation when developing multilayer Perceptron and convolutional neural networks<sup>[16]</sup>.



In addition to all that, the rectifier function is not just trivial to implement, requiring a `max()` function, but as the `max()` function is already hardwired in modern processors it is also way more efficient than the `tanh` and `sigmoid` activation function, which require the use of an exponential calculation.

Anyways, ReLU is not perfect. It has some limitations, as the key one of large weight updates having the chance of meaning that the summed input to the activation function is always negative, regardless of the input to the network (this means that a node with this problem will forever output

an activation value of 0.0). There are some alternatives to ReLU, anyways, as the Leaky ReLU (LReLU or LReL) modifies the function to allow small negative values when the input is less than zero<sup>[16]</sup>.

## How many layers do we use and how?

As everybody should know, the less layers you use on a neural network, the better performance you will get, but your result won't be satisfactory enough. To know the number of layers we should use we have to consider that just one layer won't be enough to solve this problem with a proper solution<sup>[17]</sup>.

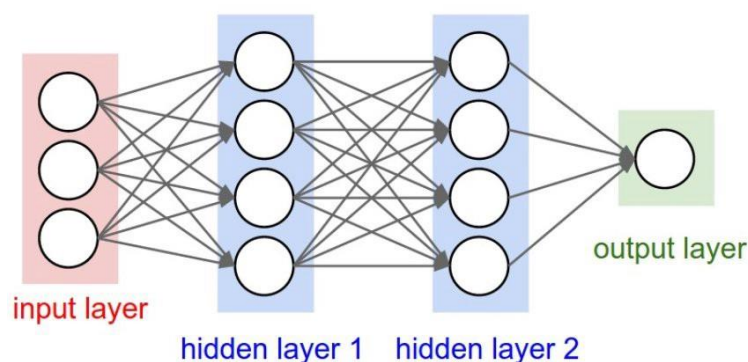
We have 5 layers for each neural network in the double deep Q algorithm: 3 convolutional in order to process the game image, then the result is flattened to be able to be fed to the next 2 regular perceptron layers that process the data into a decision in which action to perform.

For the simple deep Q algorithm only one neural network is used, but with 7 layers: again, 3 convolutional<sup>[17]</sup> layers are needed for the image processing, the output is sent to 2 other perceptron layers that work independently and then to other 2 layers that are merged at the end with a lambda function.

Both networks generate the Q values based on what they have learned and, then, are able to predict the next optimal movement.

## Sequential model

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs<sup>[18]</sup>, but it is much more simpler than functional while performing correctly the purpose for this work at learning how to play Pac-Man. As we don't want a complex model, we went for the sequential one. as the Sequential model API is enough for developing deep learning models in our situation<sup>[19]</sup>.



---

# CONCLUSION

---

At the beginning of the process, back in November, we only knew a bit about Reinforcement Learning, but for this task we have needed to learn the bases and, from there, how does Q-Learning work and its differences with other RL algorithms. We have also improved our Python and research skills, as we have had lots of problems with our code. In this matter, Internet forums and Social Networks (such as *deeplizard* in **YouTube**<sup>[20]</sup> and **StackOverflow** doubts resolutions<sup>[21]</sup>) have helped in an unimaginably way.

We used GitHub as the platform to have our project updated whenever we wanted to make any changes or tests, so everything is uploaded there. In references there will be a link to our GitHub repository<sup>[22]</sup>.

The three of us have worked at the same time, so the final evaluation for the worked hours is the same. The plan for us was to divide in the 3 of us for research and, when all of us thought we had every needed reference and knowledge for starting, we put in common everything we have found and learned. After that, we started planning the implementation of the task, searching in the Internet for help when we got stuck. Once we had a working code, we divided for training the machine using DQN and DDQN and, then, put the results in common.

It was an enriching task that made us work as a group and learn about matters we hadn't studied further from this subject. We also learned a lot about the algorithms we used and its implementation as we needed to understand the whole code in order to make the changes we needed.

*This work has been done following the **Final Task Study Guide**<sup>[23]</sup> provided by our teacher **Rafael Del Hoyo** for the subject **Intelligent Systems in San Jorge University**.*

## REFERENCES

- [1] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/REINFORCEMENT LEARNING](https://en.wikipedia.org/wiki/Reinforcement_Learning)
- [2] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/MODEL-FREE \(REINFORCEMENT LEARNING\)](https://en.wikipedia.org/wiki/Model-free_(reinforcement_learning))
- [3] [HTTPS://TOWARDSDATASCIENCE.COM/AN-OVERVIEW-OF-MONTE-CARLO-METHODS-675384EB1694](https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694)
- [4] [HTTPS://TOWARDSDATASCIENCE.COM/TD-IN-REINFORCEMENT-LEARNING-THE-EASY-WAY-F92ECFA9F3CE](https://towardsdatascience.com/td-in-reinforcement-learning-the-easy-way-f92ecfa9f3ce)
- [5] [HTTPS://MEDIUM.COM/@QEMPIL0914/ZERO-TO-ONE-DEEP-Q-LEARNING-PART1-BASIC-INTRODUCTION-AND-IMPLEMENTATION-BB7602B55A2C](https://medium.com/@qempil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c)
- [6] [HTTPS://TOWARDSDATASCIENCE.COM/DEEP-REINFORCEMENT-LEARNING-TUTORIAL-WITH-OPEN-AI-GYM-C0DE4471F368](https://towardsdatascience.com/deep-reinforcement-learning-tutorial-with-open-ai-gym-c0de4471f368)
- [7] [HTTPS://MEDIUM.COM/@QEMPIL0914/DEEP-Q-LEARNING-PART2-DOUBLE-DEEP-Q-NETWORK-DOUBLE-DQN-B8FC9212BBB2](https://medium.com/@qempil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2)
- [8] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/TABU SEARCH](https://en.wikipedia.org/wiki/Tabu_search)
- [9] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/GENETIC ALGORITHM](https://en.wikipedia.org/wiki/Genetic_algorithm)
- [10] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/A\\* SEARCH ALGORITHM](https://en.wikipedia.org/wiki/A*_search_algorithm)
- [11] [HTTPS://ELITEDATASCIENCE.COM/OVERFITTING-IN-MACHINE-LEARNING](https://elitedatascience.com/overfitting-in-machine-learning)
- [12] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/OVERFITTING#REMEDY](https://en.wikipedia.org/wiki/Overfitting#Remedy)
- [13] [HTTPS://KERAS.RSTUDIO.COM/ARTICLES/TUTORIAL OVERFIT UNDERFIT.HTML](https://keras.rstudio.com/articles/tutorial_overfit_underfit.html)
- [14] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/RETROGRADE ANALYSIS](https://en.wikipedia.org/wiki/Retrograde_analysis)
- [15] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/RECTIFIER \(NEURAL NETWORKS\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))
- [16] [HTTPS://MACHINELEARNINGMASTERY.COM/RECTIFIED-LINEAR-ACTIVATION-FUNCTION-FOR-DEEP-LEARNING-NEURAL-NETWORKS/](https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/)
- [17] [HTTPS://WWW.APRENDEMACHINELEARNING.COM/COMO-FUNCIONAN-LAS-CONVOLUTIONAL-NEURAL-NETWORKS-VISION-POR-ORDENADOR/](https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/)
- [18] [HTTPS://JOVIANLIN.IO/KERAS-MODELS-SEQUENTIAL-VS-FUNCTIONAL/](https://jovianlin.io/keras-models-sequential-vs-functional/)
- [19] [HTTPS://KERAS.IO/GETTING-STARTED/SEQUENTIAL-MODEL-GUIDE/](https://keras.io/getting-started/sequential-model-guide/)
- [20] [HTTPS://WWW.YOUTUBE.COM/CHANNEL/UC4UJ26WkCEqONNF5S26OIVw](https://www.youtube.com/channel/UC4UJ26WkCEqONNF5S26OIVw)

[21][HTTPS://STACKOVERFLOW.COM](https://stackoverflow.com)

[22][HTTPS://GITHUB.COM/LUSOR97/SISTEMASINTELIGENTESLG](https://github.com/LUSOR97/SISTEMASINTELIGENTESLG)

[23][HTTPS://PDU.USJ.ES/PLUGINFILE.PHP/63401/MOD\\_RESOURCE/CONTENT/2/GUÍA%20DE%20ESTUDIO\\_TASK\\_E\\_XAMV3.PDF](https://pdu.usj.es/pluginfile.php/63401/mod_resource/content/2/Guía%20de%20estudio_Task_E_XAMV3.pdf)